Model answer
# of AS-4159
Operating System B.tech fifth Semester
Information technology

Q.1 Objective type

Q.no	Answer
I	d(321)
Ii	C(Execute more jobs in the same time)
Iii	Three/three multiple
Iv	Max[I,j]-Allocation[I,j]
V	process
Vi	C(both a&b)
Vii	platter
viii	sector
ix	C(An illusion of extremely large main memory)
x	C(system call)

Unit 1

Ans 3: **Allocation Methods**

An allocation method refers to how disk blocks are allocated for files:

Contiguous allocation

Linked allocation

Indexed allocation

Contiguous Allocation

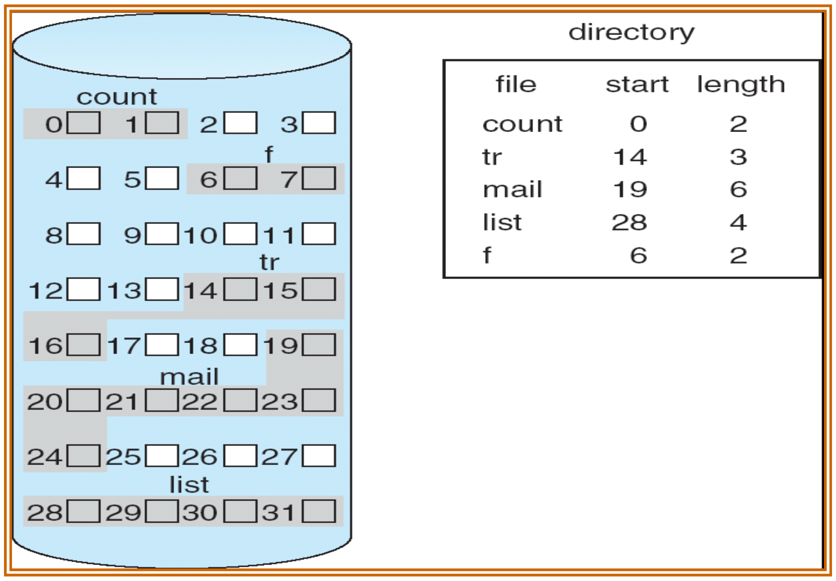
Each file occupies a set of contiguous blocks on the disk

Simple – only starting location (block #) and length (number of blocks) are required

Random access

Wasteful of space (dynamic storage-allocation problem)

Files cannot grow



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Linked Allocation

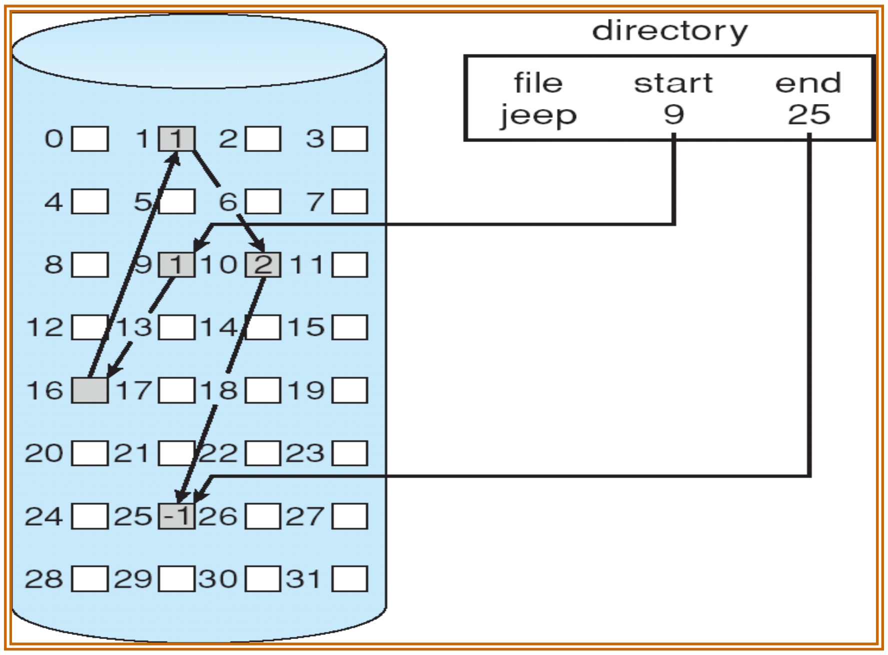
Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.

Simple – need only starting address

Free-space management system – no waste of space

No random access

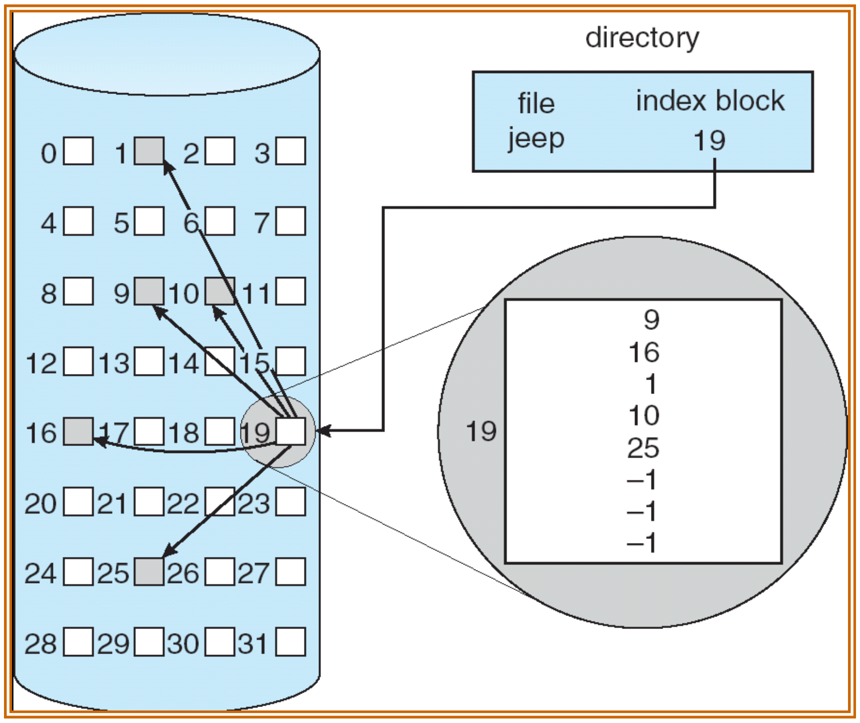
Mapping



Indexed Allocation

Brings all pointers together into the *index block*

Logical view.



Ans 4.

In the context of computing and operating systems, one might encounter many (confusing) terms which may look similar but eventually refer to different concepts. In this post, I will try to summarize the basic differences between various operating systems types and explain why and how they are not the same. The information provided is not new and can be found all over the place on the internet and actually that may add to the confusion. I hope that having all (not exactly) the terms clarified in one place would make it easy to remember. There could be many reasons why there are different operating systems types but it is mainly because each system type is designed from the ground up to meet certain design goals. On the other hand, the underlying hardware architecture influences the way systems are designed and implemented.

In a typical computing system, probably there are (many) concurrent application processes competing for (few) resources, for example the CPU. An operating system must handle resource allocation with due care. There is a popular operating systems term for such allocation known as scheduling. Depending on operating system's type, the goals of scheduling processes can be different. Regardless of system type there should be some kind of fairness when allocating resources, stated policy must be enforced and the overall system usage should be balanced. As we will see later, the goals that need to be achieved characterize the operating system type. Let us now talk about some operating systems types and terminology…

Multiprogramming

One time, I was at the post office standing in line waiting my turn to be served. My turn came but I was not fully prepared because my mail was not prepackaged. They gave me an empty box to do that on the side. I started packaging my mail while another customer is occupying my spot. It does not make sense to block the whole line while packaging my mail however it is a better idea to allow other customers proceed and get served in the mean time. I think this example (to some extent) is very similar in concept to multiprogramming model where programs are like customers and CPU is like the post office assistant. Assuming one assistant (single processor system) then only one customer can be served at a time. While a customer is being served he or she continues until he or she finishes or waits on the side. As long as the assistant is helping a customer he does not switch to serve other customers.

In a multiprogramming system there are one or more programs (processes or customers) resident in computer's main memory ready to execute. Only one program at a time gets the CPU for execution while the others are waiting their turn. The whole idea of having a multi-programmed system is to optimize system utilization (more specifically CPU time). The currently executing program gets interrupted by the operating system between tasks (for example waiting for IO, recall the mail packaging example) and transfer control to another program in line (another customer). Running program keeps executing until it voluntarily gives the CPU back or when it blocks for IO. As you can see, the design goal is very clear: processes waiting for IO should not block other processes which in turn wastes CPU time. The idea is to keep the CPU busy as long as there are processes ready to execute.

Note that in order for such a system to function properly, the operating system must be able to load multiple programs into separate partitions of the main memory and provide the required protection because the chance of one process being modified by another process is likely to happen. Other problems that need to be addressed when having multiple programs in memory is fragmentation as programs enter or leave (swapping) the main memory. Another issue that needs to be handled as well is that large programs may not fit at once in memory which can be solved by using virtual memory. In modern operating systems programs are split into equally sized chunks called pages but this is beyond the scope of this article.

In summary, Multiprogramming system allows multiple processes to reside in main memory where only one program is running. The running program keeps executing until it blocks for IO and the next program in line takes the turn for execution. The goal is to optimize CPU utilization by reducing CPU idle time. Finally, please note that the term multiprogramming is an old term because in modern operating systems the whole program is not loaded completely into the main memory.

Multiprocessing

Multiprocessing sometimes refers to executing multiple processes (programs) at the same time. This is confusing because we already have multiprogramming (defined earlier) and multitasking (will talk about it later) that are better to describe multiple processes running at the same time. Using the right terminology keeps less chance for confusion so what is multiprocessing then?

Multiprocessing refers actually to the CPU units rather than running processes. If the underlying hardware provides more than one processor then that is multiprocessing. There are many variations on the basic scheme for example having multiple cores on one die or multiple dies in one package or multiple packages in one system. In summary, multiprocessing refers to the underlying hardware (multiple CPUs, Cores) while multiprogramming refers to the software (multiple programs, processes). Note that a system can be both multi-programmed by having multiple programs running at the same time and multiprocessing by having more than one physical processor.

UNIT II

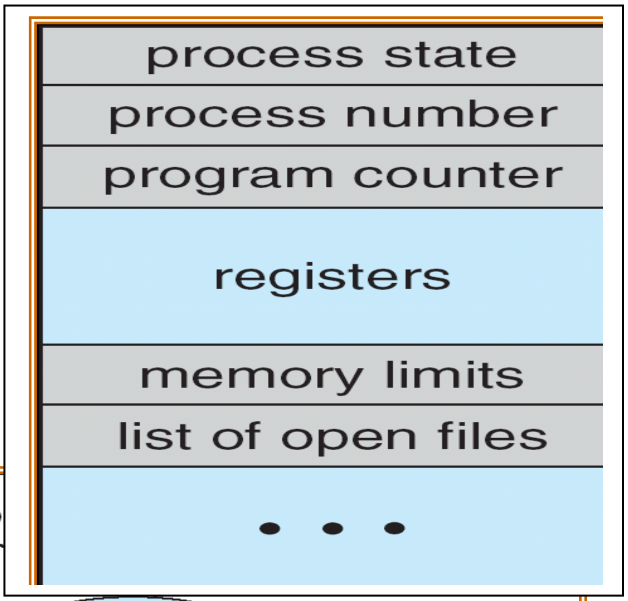
Ans 5: **Process Control Block (PCB)**

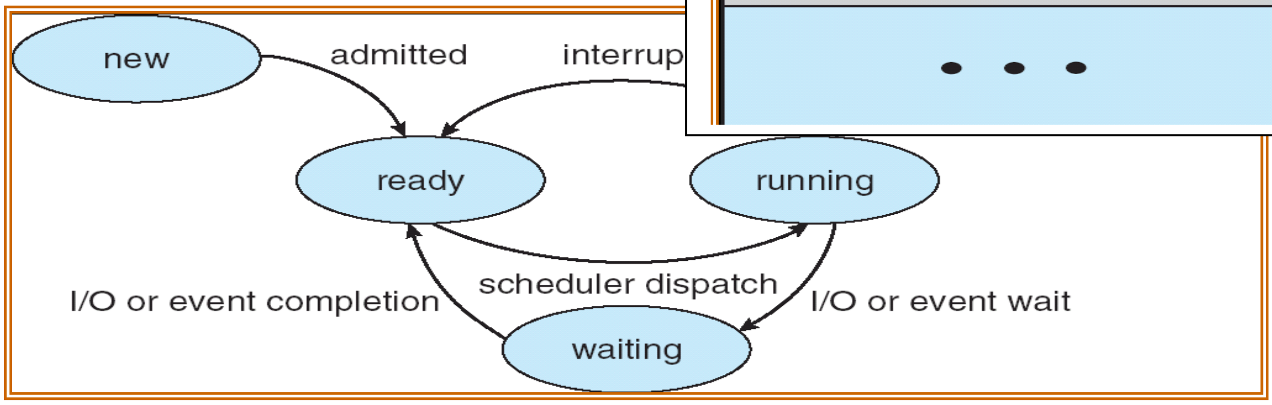
Information associated with each process

- Process state
- Program counter
- CPU registers

CPU scheduling information

- Memory-management information
- Accounting information
- I/O status information





Schedulers

Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU

ANS 6:

- *n* processes competing to use some shared data.
- No assumptions may be made about speeds or the number of CPUs.
- Each process has a code segment, called *Critical Section (CS)*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.
- When a process executes code that manipulates shared data (or resource), we say that the process is in it's Critical Section (for that shared data).
- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple processors).
- So each process must first request permission to enter its critical section.
- There are 3 requirements that must stand for a correct solution:
  - **Mutual Exclusion**
  - **Progress**
  - **Bounded Waiting**
- We can check on all three requirements in each proposed solution, even though the non-existence of each one of them is enough for an incorrect solution.

1.Mutual Exclusion – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

- Implications:
- Critical sections better be focused and short.
- Better not get into an infinite loop in there.
- If a process somehow halts/waits in its critical section, it must not interfere with other processes

2.Progress – If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely:

- If only one process wants to enter, it should be able to.
- If two or more want to enter, one of them should succeed.

3Bounded Waiting – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the *n* processes.

Unit III

Ans7:

Buffering

A buffer is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons

a. To cope with a speed mismatch between the producer and consumer of a data stream

b. To adapt between devices that have different data transfer sizes

c. To support copy semantics for application I/O

Caching

A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes.

Ans: 8

S.no	AWT	ATAT
FCFS	3.75	7.5
SGF	3.50	7.25

Unit IV

Ans9

Deadlock Conditions

1. mutual exclusion
   The resources involved must be unshareable; otherwise, the processes would not be prevented from using the resource when necessary.
2. hold and wait or partial allocation
   The processes must hold the resources they have already been allocated while waiting for other (requested) resources. If the process had to release its resources when a new resource or resources were requested, deadlock could not occur because the process would not prevent others from using resources that it controlled.
3. no pre-emption
   The processes must not have resources taken away while that resource is being used. Otherwise, deadlock could not occur since the operating system could simply take enough resources from running processes to enable any process to finish.
4. resource waiting or circular wait
   A circular chain of processes, with each process holding resources which are currently being requested by the next process in the chain, cannot exist. If it does, the cycle theorem (which states that "a cycle in the resource graph is necessary for deadlock to occur") indicated that deadlock could occur.

Banker's Algorithm

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.
- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients.

(A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)

- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.
- When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.
- The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)
    - Available[m] indicates how many resources are currently available of each type.
    - Max[n][m] indicates the maximum demand of each process of each resource.
    - Allocation[n][m] indicates the number of each resource category allocated to each process.
    - Need[n][m] indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)
- For simplification of discussions, we make the following notations / observations:
    - One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
    - A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

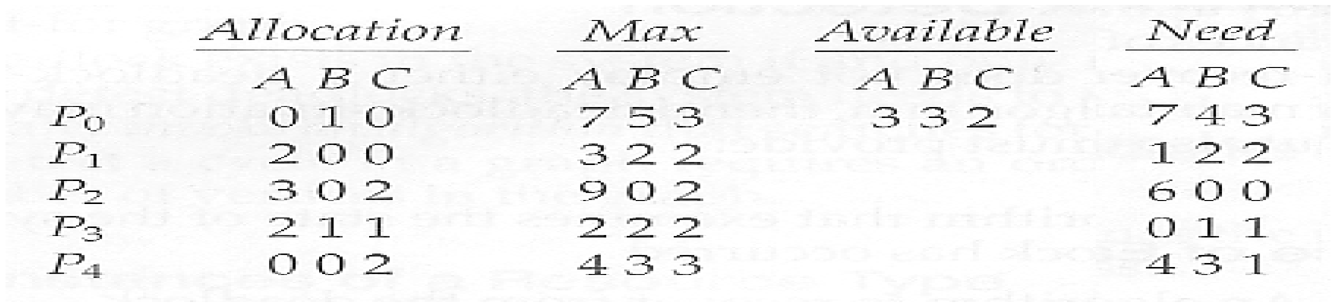
Safety Algorithm

- In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
- This algorithm determines if the current state of a system is safe, according to the following steps:
    1. Let Work and Finish be vectors of length m and n respectively.
        - Work is a working copy of the available resources, which will be modified during the analysis.
        - Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
        - Initialize Work to Available, and Finish to false for all elements.
    2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
    3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
    4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.
- (JTB's Modification:
    1. In step 1. instead of making Finish an array of booleans initialized to false, make it an array of ints initialized to 0. Also initialize an int s = 0 as a step counter.
    2. In step 2, look for Finish[i] == 0.
    3. In step 3, set Finish[i] to ++s. S is counting the number of finished processes.
    4. For step 4, the test can be either Finish[i] > 0 for all i, or s >= n. The benefit of this method is that if a safe state exists, then Finish[ ] indicates one safe sequence (of possibly many.))

Resource-Request Algorithm (The Bankers Algorithm)

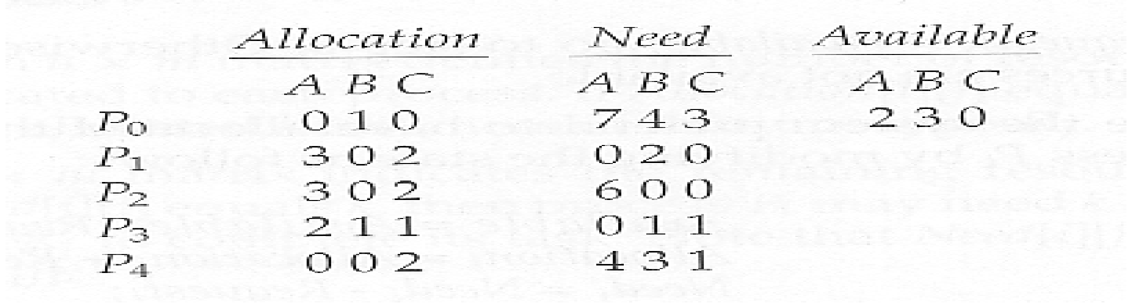
- Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
    1. Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.
    2. If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
    3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.The procedure for granting a request (or pretending to for testing purposes) is:
        - Available = Available - Request
        - Allocation = Allocation + Request
        - Need = Need - Request

Example



	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. (Request[1] = (1, 0, 2))



	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Unit V

Ans:11 the Page Fault. A page fault occurs when a program requests an address on a page that is not in the current set of memory resident pages. What happens when a page fault occurs is that the thread that experienced the page fault is put into a Wait state while the operating system finds the specific page on disk and restores it to physical memory.

When a thread attempts to reference a nonresident memory page, a hardware interrupt occurs that halts the executing program. The instruction that referenced the page fails and generates an addressing exception that generates an interrupt. There is an Interrupt Service Routine that gains control at this point and determines that the address is valid, but that the page is not resident. The OS then locates a copy of the desired page on the page file, and copies the page from disk into a free page in RAM. Once the copy has completed successfully, the OS allows the program thread to continue on. One quick note here – if the program accesses an invalid memory location due to a logic error an addressing exception similar to a page fault occurs. The same hardware interrupt is raised. It is up to the Memory Manager's Interrupt Service Routine that gets control to distinguish between the two situations.

It is also important to distinguish between hard page faults and soft page faults. Hard page faults occur when the page is not located in physical memory or a memory-mapped file created by the process (the situation we discussed above). The performance of applications will suffer when there is insufficient RAM and excessive hard page faults occur. It is imperative that hard page faults are resolved in a timely fashion so that the process of resolving the fault does not unnecessarily delay the program's execution. On the other hand, a soft page fault occurs when the page is resident elsewhere in memory. For example, the page may be in the working set of another process. Soft page faults may also occur when the page is in a transitional state because it has been removed from the working sets of the processes that were using it, or it is resident as the result of a prefetch operation.

We also need to quickly discuss the role of the system file cache and cache faults. The system file cache uses Virtual Memory Manager functions to manage application file data. The system file cache maps open files into a portion of the system virtual address range and uses the process working set memory management mechanisms to keep the most active portions of current files resident in physical memory. Cache faults are a type of page fault that occur when a program references a section of an open file that is not currently resident in physical memory. Cache faults are resolved by reading the appropriate file data from disk, or in the case of a remotely stored file – accessing it across the network. On many file servers, the system file cache is one of the leading consumers of virtual and physical memory.

Finally, when investigating page fault issues, it is important to understand whether the page faults are hard faults or soft faults. The page fault counters in Performance Monitor do not distinguish between hard and soft faults, so you have to do a little bit of work to determine the number of hard faults. To track paging, you should use the following counters: Memory\ Page Faults /sec, Memory\ Cache Faults /sec and Memory\ Page Reads /sec. The first two counters track the working sets and the file system cache. The Page Reads counter allows you to track hard page faults. If you have a high rate of page faults combined with a high rate of page reads (which also show up in the Disk counters) then you may have an issue where you have insufficient RAM given the high rate of hard faults.
Ans: 12

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process.
- Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple-partition method,
    - When a partition is free, a process is selected from the input queue and is loaded into the free partition.
    - When the process terminates, the partition becomes available for another process.
- This method is no longer in use.
- The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. In the fixed-partition scheme,
    - The OS keeps a table indicating which parts of memory are available and which are occupied.
    - Initially, all memory is available for user processes and is considered one large block of available memory, a hole.
    - When a process arrives and needs memory, we search for a hole large enough for this process.
    - If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.
- At any given time, we have a list of available block sizes and the input queue. The OS can order the input queue according to a scheduling algorithm.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- This procedure is a particular instance of the general dynamic storage-allocation problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem.
    - First fit. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
    - Best fit. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
    - Worst fit. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.
- Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.
- Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.
- External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous; storage is fragmented into a large number of small holes.
- Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.
- Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another $0.5N$ blocks will be lost to fragmentation.

- That is, one-third of memory may be unusable! This property is known as the 50-percent rule.
- Memory fragmentation can be internal as well as external.
    - Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.
    - Suppose that the next process requests 18,462 bytes.
    - If we allocate exactly the requested block, we are left with a hole of 2 bytes.
    - The difference between these two numbers is internal fragmentation; memory that is internal to a partition but is not being used.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be non-contiguous, thus allowing a process to be allocated physical memory wherever the latter is available.
- Two complementary techniques achieve this solution:
    - paging
    - segmentation
- These techniques can also be combined

Paging

- Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.
- Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store.
- The backing store also has the fragmentation problems discussed in connection with main memory, except that access is much slower, so compaction is impossible.
- Because of its advantages over earlier methods, paging in its various forms is commonly used in most OSs.
- Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and OS, especially on 64-bit microprocessors.